

U.S. PATENT APPLICATION FOR
XML TYPES IN JAVA

Inventor:

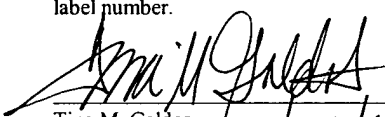
David Bau

CERTIFICATE OF MAILING BY "EXPRESS MAIL"
UNDER 37 C.F.R. § 1.10

"Express Mail" mailing label number: EV 327619127 US

Date of Mailing: 1/22/04

I hereby certify that this correspondence is being deposited with the United States Postal Service, utilizing the "Express Mail Post Office to Addressee" service addressed to **Mail Stop Patent Application, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450** and mailed on the above Date of Mailing with the above "Express Mail" mailing label number.


Tina M. Galdos

Signature Date: 1/22/04

XML TYPES IN JAVA

Inventor: David Bau

COPYRIGHT NOTICE

[0001] A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document of the patent disclosure, as it appears in the Patent and Trademark Office patent file or records, but otherwise reserves all copyright rights whatsoever.

CLAIM OF PRIORITY

[0002] This application claims priority from the following application, which is hereby incorporated by reference in its entirety:

[0003] U.S. Provisional Application No. 60/442,673, entitled XML TYPES IN JAVA, by David Bau, filed January 24, 2003 (Attorney Docket No. BEAS-01388US0 SRM/DTX).

CROSS REFERENCE TO RELATED DOCUMENTS

[0004] The following related co-pending U.S. patent applications and documents are hereby incorporated herein by reference in their entirety:

[0005] U.S. Patent Application filed _____, by Terry Lucas et al., entitled, "Programming Language Extensions for Processing XML Objects and Related Applications" (Attorney Docket No. BEAS-01235US0);

[0006] U.S. Patent Application filed _____, by Cezar C. Andrei et al., entitled, "A Declarative Specification and Engine for Non-Isomorphic Data Mapping"; and

[0007] Beas Systems Weblogic Workshop Online Help Document entitled, "Annotations Reference", <http://edocs.bea.com/workshop/docs81/doc/en/workshop/reference/tags/navJwsAnnotations.html>, version: 2003.0718.084729, dated July 18, 2003.

FIELD OF THE INVENTION

[0008] The invention relates to the transformation of data and data types.

BACKGROUND

[0009] Certain drawbacks exist in the way data is currently transformed, or marshaled, between data types. In existing systems, a user starts with an existing Java type and asks the system to generate the XML schema that reflects the Java type, and further to marshal the Java data to the XML that was automatically generated. Most products that marshal XML run through a compiler, such as a Java to WSDL compiler, in order to generate an XML schema. One drawback to such an approach is that only the scenario going from Java to XML is addressed. Current tools are not particularly good at taking an existing XML schema and turning that entire schema into a convenient-to-use Java type.

[0010] Another problem with current marshaling technologies appears when a user simply wishes to look at a small piece of XML data. That user may prefer to simply pass on the rest of the XML data without processing that data. Current marshaling technologies are not effective at simply passing on the remainder of the data. Typically, going from marshaling to unmarshaling is complicated, as not all semantics in XML can be easily captured in Java. If a user brings in a message, changes a small portion of the message, and tries to resend the message as XML, portions other than that changed by the user will be different, such that a lot of other information may be lost. If the XML contains wildcard attributes or elements, for example, those wildcards will not be retained. Information about element order may also be lost or scrambled, which is a problem if the schema is sensitive to element order.

BRIEF DESCRIPTION OF THE DRAWINGS

[0011] **Figure 1** illustrates an exemplary XML schema definition file that can be used in accordance with embodiments of the present invention.

[0012] **Figure 2** is a diagram of an exemplary system for XML marshaling and unmarshaling that can be used in accordance with embodiments of the present invention.

[0013] **Figure 3** illustrates exemplary code representing the XML types compiled from an XML schema that can be used in accordance with embodiments of the present invention.

[0014] **Figure 4** illustrates exemplary code of an annotation-based format that can be used in accordance with embodiments of the present invention.

[0015] **Figure 5** illustrates exemplary code implementing a Web service that can be used in accordance with embodiments of the present invention.

[0016] Figure 6 illustrates exemplary code for type transformation that can be used in accordance with embodiments of the present invention.

[0017] Figure 7 illustrates exemplary code for default type declaration that can be used in accordance with embodiments of the present invention.

[0018] Figure 8 illustrates exemplary code for XML transformation that can be used in accordance with embodiments of the present invention.

[0019] Figure 9 illustrates exemplary code for XML transformation that can be used in accordance with embodiments of the present invention.

[0020] Figure 10 is a diagram of an exemplary XML transformation system that can be used in accordance with embodiments of the present invention.

[0021] Figure 11 is a diagram of an exemplary system for XML store that can be used in accordance with embodiments of the present invention.

[0022] Figure 12 is a diagram of an exemplary system for XML schemas that can be used in accordance with embodiments of the present invention.

[0023] Figure 13 is a diagram of an exemplary system for XML types that can be used in accordance with embodiments of the present invention.

[0024] Figure 14 is a diagram showing an exemplary hierarchy diagram that can be used in accordance with embodiments of the present invention.

DETAILED DESCRIPTION

[0025] The invention is illustrated by way of example and not by way of limitation in the figures of the accompanying drawings in which like references indicate similar elements. It should be noted that references to “an” or “one” embodiment in this disclosure are not necessarily to the same embodiment, and such references mean at least one.

[0026] Systems and methods in accordance with one embodiment of the present invention overcome many deficiencies in existing marshaling and unmarshaling systems by translating XML schemas, which define XML data in an XML document, into XML types in Java when marshaling data between XML and Java. XML types are actually Java types which, in addition to regular Java bean functions and access to database, can also access and update XML data within Java in an efficient, type-safe, robust, and convenient way. An

architecture can be used in at least one embodiment that provides the ability to handle almost 100% of the schema introduced by a user.

[0027] The use of XML types can allow the combination of XML- and Java-type systems. This can be done in a way that allows developers to achieve loose coupling. XML schemas realized as XML types can remain fully faithful to the XML. It can be easy for a developer to take control of precise transformations between existing Java types and existing XML types. XML types can address the Java/XML boundary by bringing together several technologies, including for example schema-aware strongly typed access to XML data document from Java, compact and indexed in-memory XML store, speedy and minimal (pull) parsing and (binary) serialization, lightweight document-cursor traversal, XPath and XQuery navigation and transformation, and seamless Java IDE integration

[0028] For an XML-oriented “XML to XML via Java code” example, **Figure 1** shows a simple exemplary XML schema definition (XSD) file that can be used with embodiments of the present invention. This particular XML schema describes the type of a purchase order. The schema can be a pre-existing file that was generated by a schema tool or created by a user. In this example, it may be necessary to “clean up” or “fix” invalid XML purchase order information.

[0029] At the top of the XSD is a schema definition for an element named ‘purchase order.’ The element is defined as a complex type and contains an element called “line-item.” There is a ‘maxOccurs’ attribute set to ‘unbounded,’ meaning that the line-item can be repeated any number of times. Each of the line items is shown to have four sub-elements: desc, itemid, price, and qty. The ‘desc’ sub-element refers to a description element which is a string, ‘itemid’ refers to an item identifier element which is an integer (type int), ‘price’ refers to a price element which is a floating point number (type float), and ‘qty’ refers to a quantity element which is an integer. These sub-elements are all built-in types of the schema. This schema is basically a description or representation of how a valid purchase order schema should look.

[0030] In order to write a program using XML types, an XML schema file can be added to a Java project. An example of a system for XML marshaling and unmarshaling is shown in **Figure 2**. In one embodiment, the system can process the file with a compiler **100** that knows not only how to compile a Java project **101**, but also is capable of compiling the XSD file **102**. When XSD file is compiled, a number of XML types **103** can be generated in addition to regular Java types **104**. These XML types can then be added to the classpath.

For example, an XML type called “purchase order” can be generated from the schema type called “purchase-order.”

[0031] A Java source code representation of the XML types compiled from the example schema in **Figure 1** is shown in **Figure 3**, where the source that generates these Java types is the schema file itself. A type called LineItem corresponds to the line item-element nested inside the purchase-order element in the XSD file. In Java, the XML line-item element can turn into something similar. For each element inside a type, a Java field can be generated. For the “desc” element in the XSD, for instance, there are corresponding “getDesc” and “setDesc” methods in the generated type for each line item. For the purchase order as a whole, a user can obtain or send an individual line item. The names of the generated types can be automatically derived from the schema names. Each generated type can be annotated with the relevant schema name from which it comes. Each type can also extend an existing XML type.

[0032] In one embodiment, XML types can implement a common base XML type called “XMLObject”. Such an XML type provides the ability to execute a number of XML-oriented data manipulations and will be referred to herein as an “XBean”. An XBean is not a standard Java bean, as an XBean inherits from an XMLObject. An XMLObject is unusual, in that an XMLObject as an XML type provides a way for each XBean to retrieve its original, or corresponding, XML. An XBean can be thought of as a design pattern for Java types representing data, such as business data, that can be serialized and de-serialized to XML, as well as accessed in a type-safe way from Java. XBeans can also be thought of as a small set of natural language idioms, either annotated Java or non-Java, for generating those types. Normally, there is a tradeoff when an application developer or component developer decides how to represent business data. If the data is represented as type-safe Java, then serialization to XML or to databases can be awkward. If the data is represented as XML, then conversion to Java types can also be somewhat awkward. The same holds true for conversion to either of the other types if data is in a result set from a database. It is therefore advantageous to provide a single category of Java types that is convenient for passing, using, and manipulating as both XML and Java. It is further advantageous that the same type is convenient for database access as well as form input and validation.

[0033] As Shown in **Figure 2**, XBeans in one embodiment can sit at the intersection of three types of business data: XML data 105, Java data 106, and database 107. An XBean can simply contain data, without any logic or means for communication or computation. An XBean can be defined using any of a number of

idioms. Once an XBean is defined, the XBean can be used in various contexts. For example, an XBean can be defined using XML schema, then instantiated by attaching the XBean to an XML input stream. In this case, the Java types that are generated can be passed around and used as a Java bean, with friendly getters and setters.

[0034] For example, an application developer could define a file called MyData.schemabean, with the following contents:

```
<xsd:element name="myData">
  <xsd:sequence>
    <xsd:element name="a" type="xsd:string">
    <xsd:element name="i" type="xsd:int">
  </xsd:sequence>
</xsd:element>
```

This file could compile into a XBean with metadata, such as in **Figure 4**, which shows an exemplary annotation-based format, which would be another way of expressing an XBean.

[0035] If the data was XML-oriented, for example, a user might have cared whether a name came before a description or a description came before a name. Systems and methods in accordance with the present invention allow a user to get back to the original, ordered XML. There can be any of several methods on a base XMLObject type, such as methods called “getXMLCursor” and “executeXPath.” An XML type can have a number of XML data operations, including methods to query values using XPath, transform values using XQuery, or iterate over the data document using an XMLCursor. This base type can hold several technologies together.

[0036] Certain methods can determine what the XML looks like at any point in time. For example, XML types can be used to implement a Web service that executed the requested operation such as shown in **Figure 5**. The exemplary code in this Figure can be used to fix quantities in an entire purchase order. On lines 1-2 of the Figure, the method is declared as a Web service operation that takes a PurchaseOrder XML type. The Web services runtime can recognize XML types and pass incoming messages efficiently, without fully parsing the XML. The other XML type that was declared in the schema is LineItem, which can be seen on lines 4, 5, and 7 as an array called emptyItems. Since the compiled XML types are strongly typed, they allow validation of both schema, such as validating PurchaseOrder against a schema file, and Java compiler checking, such as verifying that the type of the argument of emptyItems[i].setQty(1) is an integer.

[0037] For instance, a user might receive a number of purchase orders with high line items that have erroneously set the line item with quantity 0. That user may wish to manipulate these purchase orders, such that whenever somebody leaves a quantity field set to 0, the system changes that quantity field to 1. A function can take a purchase order XBean as input, and an XPath can be executed on the purchase order which looks for any line item tags underneath the purchase order tag that have quantity equal to 0. This is shown, for example, starting at line 5 in the Figure. A list of nodes can be returned to the user that match the XPath. That list of nodes can be cast back to the XBean type that the user knows it to be. The system selects a set of line items that can be cast to an array. Once the results are obtained from XPath, it is possible to iterate through the results and use an XBean method to manipulate the result nodes.

[0038] Strongly typed Java accessors may not be appropriate for all XML usage. In one embodiment, XML types can extend a base XML-oriented XMLObject type that provides, for example, XPath, XQuery, XMLCursor, and XMLReader. On line 5, the XMLObject getAllValues method executes an XPath on the input to locate all line item elements with qty=0. On lines 6 and 7, it can be seen that the LineItem types can be used to update the XML data document. Each instance of the type refers to a specific node in the document, and when methods such as setQty(1) are called, the data of the document are being manipulated in an easy, type-safe way. On line 8, the type is returned directly from the Web service to complete the function and send the response message.

[0039] Systems and methods in accordance with embodiments of the present invention can also deal with transformation among different XML types, where a user may need to process an XBean to retrieve data. For example, it may be necessary to clean up the line items by modifying the description and price to match the item ID. This can be done in one example by looking up each catalog item in an existing application database. This work can be done using a Java lookup method that can take an integer item id and return a CatalogItem type. For instance:

```
CatalogItem findCatalogItem(int catalogID);  
class CatalogItem  
{  
    int getCatalogID();  
    String getDescription();  
    float getPrice();  
}
```

This class appears to be similar to a LineItem XML type, but has some relatively minor differences. For instance, CatalogItem has no quantity and the item ID is called a “catalogID” rather than an “itemID.” In fact, since CatalogItem is so similar to LineItem, it may be desirable in some situations to write code such as that shown in **Figure 6**. In the Figure the value of a complex XML type, LineItems, is being set to a complex Java type, CatalogItem. Each XML type has a set method that can take an arbitrary type, such that the above code can compile and run. At runtime, however, the user can get an XML conversion exception, complaining “No Transformation has been defined mapping from CatalogItem to LineItem”. In such case, the user can select the appropriate CatalogItem type name and check the corresponding XML transformation. If a user has an existing bean that looks up the title of an item and returns a Java type called ‘CatalogItem,’ CatalogItem will have a CatalogID description that looks like the line items, but is slightly different than the prior example. For example, there is no quantity field, and what was previously called “ItemID” is now called CatalogID.

[0040] In systems and methods in accordance with embodiments of the present invention, each XML transformation can be implemented using an XQuery stored in a directory of transformations. Each XQuery can transform from one or more types, each with a known XML schema, into a specific type. A visual XQuery editor can be used that has input and output types pinned to the line-item type, as well as the default type for CatalogItem. An XQuery editor can allow a system or user to connect itemID to catalogID, and to indicate that the quantity should be zero.

[0041] An example of a default type declaration for the CatalogItem class is shown in **Figure 7**. It can be visually written by XQuery into a file in a directory that contains all the transformations. An XQuery can be defined that tells the system how to transform a catalog item into a regular line item. An example of such a query is shown in **Figure 8**. The bolded text in **Figure 8** corresponds to an XQuery that maps catalogID into itemID and sets the extra field qty to 0. The remainder is an envelope that holds the XQuery in an XML file. This 0 value can be corrected, such as can be done in Java as shown in **Figure 9**. On line 13 in **Figure 9**, exemplary code can be seen where the catalog item is being sent into the items line. It can be desirable to take a catalog item that a user is getting from an existing API and get that item into an XBean that will represent a line item. Such code would not normally work, as the catalog item is different from the line item.

[0042] An example of an XML transformation system is shown in **Figure 10**. In one embodiment, a global registry of transformations(XQueries) 111 can be used to get from one data type to another. Once a set of

XQueries is obtained, the transformation system 110 can automatically look up whether a transformation exists from a source type 108 to a target type 109. In another embodiment, a library of transformations 112 can still be used, but instead of automatically transforming from a source type to a destination type, each transformation will be given a name. When a catalog item is given and a regular line item needs to be set, for example, the user can simply invoke the transformation by name before the set is called. In **Figure 9** where lines 13 and 14 are in bold type, there will be a function call between the 'set' and 'findCatalogItems' that tells the system to transform the catalog item to a line item. One reason to make this transformation explicit is that an invisible registry of transformations can be too abstract and "under the covers" for many users. If a user has data transformed, that user may want to actually see the name of the transformation in the code. If a user can see the name, it is possible for the user to navigate through the code to determine what the transformation does and how it affects the data. This can act as a check so the user can be sure that the transformation is understood, whether the code belongs to the user or someone else.

[0043] An alternative way to ensure that the quantity is correct is to define the CatalogItem through a line-item transformation to take two input arguments, such as a CatalogItem and an integer quantity. From this example, it can be seen that there is a global registry of transformations, indicating source types and target types. Sources and targets are allowed to be Java types. Whenever an automatic translation between two different types is required, the registry can be consulted. A registry can be used that allows a single Java type to map to any number of different XML types, depending on the situation. A registry can also have the advantage that every mapping between any two given types need only be defined once, and then it is easily reusable.

[0044] In certain systems, difficulties may arise such that multiple versions of a schema may need to be dealt with at the same time in a single program. For this reason, there can be a provision for tagging each schema with a version identifier. The relevant Java types and transformations can all be done separately, treating each version as its own type system.

[0045] In yet another example, a user may wish to write a Web service that takes a catalog item as input, or to expose an existing Java function such as "findCatalogItem" as a Web service. For example, the following code could be written to expose findCatalogItem as a Web service in existing systems:

```
class MyWebService
{
    /** @jws:operation */
```

```

CatalogItem findCatalogItem(int catalogID)
{
    return MyDbUtilities.findCatalogItem(catalogID);
}

```

[0046] Such an approach may be acceptable where a user is defining the proper WSDL type for that user's Web service. Unfortunately, the situation may be such that there is an existing XML schema describing the desired result type. For example, the results may be returned as a standard purchase order line-item element as in the examples above. In such case, the actual WSDL may not conform to the existing XML schema. An attempt can be made to create the proper schema using XML transformation, but other than providing a convenient syntax, existing systems provide no assistance in ensuring that the map conforms to the schema. Using a transformation registry in accordance with various embodiments of the present invention, however, it can be easy to ensure that the return result conforms to the proper type.

[0047] For example, the code could be modified as follows:

```

class MyWebService
{
    /**
     * @jws:operation
     * @jws:return-schema type="po:line-item"
     */
    CatalogItem findCatalogItem(int catalogID)
    {
        return MyDbUtilities.findCatalogItem(catalogID);
    }
}

```

The code above requires that there be a defined XML transformation that maps the CatalogItem type to the type of the po:line-item element. If there is none, the IDE will signal an error on the return-schema annotation. However, if there is a defined transformation, the return type of the Web service method can conform to the requested schema and the necessary schema can be included in the emitted WSDL.

[0048] In the example, such as can be seen in the line including "return-schema type=po:line-item," the system allows a user to return a catalog item, but have the line item in the XML. The system can go to a registry of XQueries and execute the instructions using an explicit XQuery. This embodiment provides for the association of an XQuery with an XML Map, the support of every schema, and a way to get simultaneous access to both the strongly-typed view and the XML view of the data.

[0049] As Shown in **Figure 11**, when systems and methods in accordance with one embodiment load a piece of XML as an XML type 114, that XML type can be loaded into a lightweight internal XML store 115. This is similar to what can be done with the W3C Document Object Model (DOM), but is done in a more lightweight way, which can have complete fidelity to the original XML. The store is lightweight, since it only retains the XML data 113 either as a searchable index, or as in the current example, at approximately the text or tag level. Therefore, when a user has an XML type such as a purchase order, the user can simply have a handle to a location of one of these lightweight XML stores, which can resemble XML trees. When the user obtains the reference, there may not yet have been any marshaling. A getter can be called when a user calls a method such as “getLineItem,” “getLineItem.getDesc,” or “getQty,” the getter being used to marshal, determine the type, and return the answer. This approach can be very time efficient, as only a portion of the data is being processed.

[0050] Systems and methods in accordance with some embodiments can keep an XML schema and a corresponding Java type in sync. A user with a strongly-typed Java can begin to add new line items or to change quantities, for example. If that user then wants to run an XPath path on the Java type, the XPath may need to be run on the XML data document in the current form of the data. In this case, if a user makes a modification to a document, either on the XML side or on the strongly-typed Java side, the appropriate portion of the other side can be invalidated. When a user subsequently looks at that other side, the previously-invalidated information can be faulted in.

[0051] In order to compile an XML schema, it can be necessary to parse the schema, or XSD file, which is referred to as “schema for schema”, In other words, an XSD file that represents the appropriate schema for the XSD files themselves. If a system is supposed to be able to handle 100% of the schema passed to the system, and the system generates convenient Java accessibility, it can be expected that the system uses its own generated types for understanding XSD files when the system reads schema. A user should be able to take the schema for schema and compile that into Java, such that the system can simply use the Java.

[0052] Systems and methods in accordance with some embodiments, as shown in **Figure 12**, there can be at least two parts to any piece of data, or XML data, including the legal type of the data 116 and the meaning of the raw data 119. The legal type of the data defines what kind of data is regarded as valid for the current application. A schema can contain very specific details regarding the legal types of data, and can in some instances contain some detail regarding the meaning of the data. Once the legal type of the data is known,

it is possible to generate an automatic type that provides access to that data in a strongly-typed way. This is possible in part because the schema can identify all the valid sub-fields of a given data field. It is then possible to grant a user strongly-typed Java access in the appropriate place(s). Even after a user has loaded the data and has the data in the appropriate type, the user may still not know what the data means. In such case, a schema compiler 117 can be used that understands the raw data. This is somewhat similar to what are known as compiler compilers, such as YACC (“Yet Another Compiler Compiler”), which are capable of taking an abstract grammar and compiling the abstract grammar into a syntax tree. Since XML is already a syntax tree, this is not a problem. XML is not, however, a constrained syntax tree. Any node can have a variety of elements beneath it in the tree. Once a user has a schema, which can be thought of as a grammar for XML, the user knows exactly what is supposed to be underneath any given node. Therefore, instead of using just a raw XML syntax tree, a user can take advantage of a schema-constrained syntax tree.

[0053] Systems and methods in accordance with one embodiment of the invention maintain each schema as a Java type, including simple types. If a user has a schema that is a restriction of a simple type, it can be indicated in the schema. For instance, if a user-defined type to be an integer of a legal type, it has to be a five digit number between 10,000 - 99,000. It is not necessarily desirable to define this to be a simple integer type as in existing systems. Instead, the information can be generated into a Java type. The name of the Java type can then be generated from the schema, such as the name “restricted integer.”

[0054] Another invariant that can be maintained by systems and methods in accordance with the present invention arises in cases where there are at least two types in a schema that are base types. If one of the types is a base type of the other, that relationship will connect the two types in Java. A high-fidelity translation of typed systems can allow base types to be preserved.

[0055] A validation engine using complied XML type constraints 118 can also be used to allow a user to determine whether any relevant XML type 120 is valid according to the XML. For example, in XML a purchase order line item might have a description quantity, catalog number, and a price. There may also be a restriction in the appropriate XML schema that indicates ‘description’ is optional, but ‘catalogItemNumber’ is not optional. In Java, there is no way of indicating that a field is not optional, or cannot be null. As such, most people who do marshaling are not able to validate a bean. Validate methods in accordance with embodiments of the present invention can be used that allow a use to validate any bean against the XML type constraints, and to be informed of any validity problems.

[0056] In systems and methods in accordance with some embodiments, an XML type can be shared among multiple Java components. An XBean can be automatically emitted, such as where an automatically generated XML type is defined for a user-defined component works. In such a case, XBeans representing parameters and return values can be auto-generated as inner classes to an XML control interface generated for the component. If the message types are actually shared across many components, it may not make sense to have private XBean types for each instance of the message. In such case, it should be possible to refer to an XBean type explicitly when defining a user-defined component, in order to explicitly control how the XML type of the component is shaped. For example:

```
package mypackage;

/**
 * @jws:xml-interface enable="true"
 */
class MyComponent
{
    /**
     * @jws:operation
     * parameter-xml-type="MyData"
     * return-xml-type="MyStringMessage"
     */
    String myOperation(String a, int i) { return a + i; }
}
```

By referencing the XBean type "MyData" in the component, such as for parameter-xml, it can be asserted that the bean has getters that correspond to the argument names. For example, getA should return a String, and getI should return an int. If these types do not line up, it may result in a compile-time error. For return-xml, it can be asserted that the bean type is the return value, or that it has a single property whose type matches the return value. By referencing the XML type, the XML schema is being referenced that defines the type of input and output messages to this method. The schemas can be reusable since they have names such as "MyData". A map is also being referenced between the XML and the Java types. The map can be attached to the MyData type as metadata and, since it is attached to a named type, the map can be reusable.

[0057] A generated XML control interface that can be obtained when specifying explicit XBeans on the component can be as follows:

```

package mypackage;

interface MyComponentXMLControl
{
    /**
     * @jws:return-xml xml-type="MyStringResult"
     * @jws:parameter-xml xml-type="MyData"
     */
    XML myOperation(XML x);
}

```

In this example, the named XBean types are used to specify the xml schemas allowed, and there are no generated inner classes.

[0058] An XBean type can extend a base XML type, such that wherever XML can be passed, an XBean can be passed as well. In addition, any XBean can be attached to an XML data document, so wherever XML is available, an XBean can be created for convenient access to the data. In systems and methods in accordance with some embodiments, XBeans can be created easily and used in several different ways. For instance, an XBean can be created implicitly via the definition of a JWS (Java Web services) method. An XBean can be created based on a parameter list of the function and the maps associated with the function. Also, an XBean can be created explicitly using a *.xbean file. A *.xbean file can have at least two different implementations, such as JavaBean+Maps or XML+Query, each of which can freely use annotations. An example of implicitly creating a bean over a JWS operation might look like the following:

```

/**
 * @jws:operation
 * @jws:usebean beanName::creditCardInfo::
 */
updateCreditCardInfo(String custId, String ccNumber, Date expDate, Address addr)

```

This would implicitly create an XBean using the default maps for the operation. Applying a specific map to the operation would create the XBean using those maps for input and output. If the beanName is already defined, the existing bean can be used. A separate syntax can be used when creating a bean, instead of using an existing syntax.

[0059] A simple JavaBean+Maps *.xbean file might look like the following:

```

public interface xbean1
{
    /**
     * @xbean:property
     */
    String name;
}

```

```

/**
 * @xbean:property
 * */
String address;
}

```

This would create a file with public get and set methods, as well as the standard XML that would be defined for this set of properties. A slightly more complex file would have a map attached, such as:

```

public interface xbean1 extends JavaXBean
{
/**
 * @jws:inputxml xml-map::
 * <PERSON><NAME>{name}</NAME><ADDRESS>{address}</ADDRESS>::
 **/

/**
 * @xbean:property
 * */
String name;
/**
 * @xbean:property
 * */
String address;
}

```

The above examples use individual Java members as native storage. Equally important can be the use of XML as a native storage. On the opposite side, a simple file could use XQuery to return values. This might look as follows:

```

public interface xbean1 extends XMLXBean
{
    private XML xml;

    void xbean1(XML xmlParam)
    {
        xml = xmlParam;
    }

/**
 * @jws:xquery statement :: $xml/name/text() ::
 * */
String getName();
}

```

[0060] In systems and methods in accordance with some embodiments, there can be two XML types: movable cursors and immovable values from a developer's point of view. As cursors are moved, the part of the XML data document viewed by it can change, so the types can be designed to operate anywhere within

an XML data document. On the other hand, immovable values can be fixed in one place, so they can have strongly-typed methods that match the XML schema of the part of the data document that they reference. **Figure 13** shows an example of such an implementation, wherein XMLIndex 121 represents immovable value while XMLCursor 122 or 123 represents movable cursor of the underlying XML data. The actual implementation of a strong XML type may not use an XMLCursor, but it should be noted that the purchase order interface 124 provides the same kind of reference into underlying XMLIndex , even though it is immobile.

[0061] Although a user can use and manipulate a strongly type such as PurchaseOrder as if it were an ordinary Java type, behind the type can be an implementation that directly accesses and manipulates the underlying XML data. For example, immediately after a value is set in a strongly type, the same value can be available from any cursor that uses XPath to search the same set of data.

[0062] The model shown in **Figure 13** can be useful when an XML data document is being held, queried, manipulated, and reused. At other times, it can be necessary to stream XML for one-time-use without ever holding on to the data document. For those situations there can be an XMLReader class such as an XML pull parser. Each XML Value object can stream itself out by supplying an XMLReader. There can also be an XMLIndex constructor that loads and indexes the contents of an XmlReader.

[0063] **Figure 14** shows an exemplary class hierarchy diagram for an XML type API. In the Figure, the rounded boxes designate interfaces while the squared boxes designate concrete classes. A user starting from a file or a raw input stream can parse the XML using an XML Parser, then index the parsed file using an XML Index. These are the only two concrete classes shown, and these classes are factories for XML types that implement all the other interfaces.

[0064] XML types can add the schema to the Java runtime model. For example, every schema can compile into a Java type at compile time. This can include both complex types and simple types. Precompiled types such as XmlString and XmlDate can be used for the fundamental and simple types built-in to XML Schema. XMLObject itself can correspond to the xsd:anyType. In addition, for each schema, a pointer resource can be generated into the target class hierarchy that provides a map from all schemas with a given name into corresponding java type names.

[0065] Multiple schemas can be allowed to have the same XML name, but different types with the same name may be tagged with different “XML world” names. Only one world may be allowed to be the default

world. One way to control type generation is through an .xval file adjacent to the .xsd file at compile time. At runtime, indexed XML can be automatically schema-aware. The visibility of schemas can be tied to the current ClassLoader. A thread-local index of visible schemas can be maintained. When a new schema is requested via fully-qualified XML name, a ClassLoader.getResourceAsStream call can be used to locate a pointer to the corresponding Java type, such as in the default world. Lookups in a specific world can also be done. An implementation of XMLIndex can automatically resolve all XML to types using such a scheme. If no "xml world" is specified, a default world can be used. Other alternate views can also be specified that allow different versions of schemas to be used.

[0066] One embodiment may be implemented using a conventional general purpose or a specialized digital computer or microprocessor(s) programmed according to the teachings of the present disclosure, as will be apparent to those skilled in the computer art. Appropriate software coding can readily be prepared by skilled programmers based on the teachings of the present disclosure, as will be apparent to those skilled in the software art. The invention may also be implemented by the preparation of integrated circuits or by interconnecting an appropriate network of conventional component circuits, as will be readily apparent to those skilled in the art.

[0067] One embodiment includes a computer program product which is a storage medium (media) having instructions stored thereon/in which can be used to program a computer to perform any of the features presented herein. The storage medium can include, but is not limited to, any type of disk including floppy disks, optical discs, DVD, CD-ROMs, micro drive, and magneto-optical disks, ROMs, RAMs, EPROMs, EEPROMs, DRAMs, VRAMs, flash memory devices, magnetic or optical cards, nanosystems (including molecular memory ICs), or any type of media or device suitable for storing instructions and/or data.

[0068] Stored on any one of the computer readable medium (media), the present invention includes software for controlling both the hardware of the general purpose/specialized computer or microprocessor, and for enabling the computer or microprocessor to interact with a human user or other mechanism utilizing the results of the present invention. Such software may include, but is not limited to, device drivers, operating systems, execution environments/containers, and applications.

[0069] The foregoing description of the preferred embodiments of the present invention has been provided for the purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise forms disclosed. Many modifications and variations will be apparent to the practitioner skilled in

the art. Particularly, while the concept “type” is used for both XML and Java in the embodiments of the systems and methods described above, it will be evident that such concept can be interchangeably used with equivalent concepts such as, interface, shape, class, object, bean, and other suitable concepts. Embodiments were chosen and described in order to best describe the principles of the invention and its practical application, thereby enabling others skilled in the art to understand the invention, the various embodiments and with various modifications that are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the following claims and their equivalents.